Time and Energy Profiling in Production Sensor Networks with Quanto

Basil Crow

Department of Computer Science

Brown University

Abstract

We present improvements to Quanto, a network-wide time and energy profiler for embedded network devices. We present an efficient method for estimating the individual energy consumption of concurrently operating hardware components when only their aggregate energy is observable. We also present a method for causally connecting energy usage to high-level, programmer-defined activities without recourse to bandwidth-intensive logging. Our improvements drastically reduce the impact of Quanto's time and energy profiling on system behavior, enabling Quanto to scale from test environments to production sensor networks alike.

1 Introduction

Energy is a scarce resource in battery-operated embedded devices, such as those used in wireless sensor networks. This scarcity has motivated a wide variety of research in such areas as platform design, networking protocols, operating system abstractions, middleware protocols, and data aggregations. Despite the abundance of research in these areas, however, few of the claims made in the literature have been justified using empirical metrics and realistic workloads [7]. The lack of such evaluation in the literature is problematic not only because it casts a cloud of uncertainty over the correctness and efficiency of the work being presented, but also because the energy consumption in production sensor networks often inexplicably differs from expectations or what lab tests suggest [8]. To make matters worse, the infrastructure for time and energy profiling in production sensor networks is generally poor or nonexistent.

Addressing these problems requires a thorough understanding of how and why embedded applications spend energy and has prompted numerous questions. For example, how much energy do individual operations, such as sampling sensors or receiving packets, cost? What is the energy breakdown of a node in terms of activity, hardware, and time? Network-wide, how much energy do network services, such as routing or time synchronization, consume?

Several factors make these questions difficult to answer. For example, nodes have limited processing power and storage, with clock speeds on the order of tens of megahertz and storage capacities on the order of kilobytes of RAM. To be acceptable for use in production sensor networks, profile collection must have a negligible impact on the operation of the system as a whole. In addition, a profiling system must tie together separate operations across multiple energy consumers, such as sampling sensors, sending packets, and CPU operations, in order to highlight systemic trends.

These challenges motivated the development of Quanto [5], a network-wide time and energy profiler for embedded devices. Quanto is implemented for TinyOS, a popular operating system for sensor networks, and has two main features. First, Quanto answers the question *where have all the joules gone?* by estimating the individual energy consumption of concurrently operating hardware components when only their aggregate energy is observable, leveraging an energy sensor based on a simple switching regulator [2] to take fine-grained measurements of energy usage as cheaply as reading a counter. Second, Quanto answers the question *why were those joules spent?* by causally connecting energy usage to high-level, programmer-defined activities.

Since its release in 2008, Quanto has been used in several case studies. For example, Quanto was used to investigate the interference from an 802.11 b/g network on the operation of low-power listening, a family of duty-cycle regimes for wireless radios, as well as to uncover a previously undetected timer-related bug in TinyOS [5]. More recently, it was used to characterize the power draw of a wireless sensor network running the Collection Tree Protocol (CTP) as a function of low-power-listening interval [7]. On the one hand, these case studies have demonstrated the valuable insights into device and network behavior provided by Quanto through time and energy profiling. On the other hand, however, they have brought to light the lack of scalability of the original Quanto prototype, both in terms of estimating the energy breakdown of hardware components as well as tracking activities. Below, we discuss these scalability problems, present our improvements, and evaluate their efficacy in production sensor networks.

2 Background

2.1 Energy tracking

An embedded device consists of a set of concurrently operating hardware components, such as the CPU, radio, and LEDs. Each of these components consists of functional units, which we call *resources,* and each resource has operating modes with different power draws, which we call *power states*. A power state consists of a set of *power state bits*, each of which is a binary variable that tracks whether some specific component of the device is active or inactive. At any given time, the aggregate power draw for a system is determined by the set of power states of its resources. For more information about the hardware platform, resources, and power states used by Quanto, see §§2.2–2.3 of [5].

Suppose that the total number of power state bits over all resources in the system is n. In

the original Quanto prototype, device drivers are modified so that they intercep all events that changed the power state of a resource. Any two such events form an interval i, and for each interval the operating system tracks the aggregate energy consumed during the interval (ΔE_i), the length of the interval (Δt_i), and the setting of each power state bit — active or inactive — during the interval ($\alpha_{i,1}, \ldots, \alpha_{i,n}$), logging this information to the serial port. Once this information is received, we consolidate all intervals that have the same global power state g (a particular setting of $\alpha_{i,1}, \ldots, \alpha_{i,n}$) into a single interval whose length and aggregate energy spend is the sum of the corresponding metrics of each of its constituent intervals. We refer to the total number of intervals after consolidation by m, which is at most 2^n.

Based on this information, Quanto's offline processing tool generates one linear equation of the following form for each interval:

$$\Delta E_i = \Delta t_i \sum_{j=0}^{n} \alpha_{i,j} p_{i,j},$$

where $p_{i,j}$ is the (unknown) power draw of the j-th power state bit during the i-th interval. The average power over the interval $P_i = \Delta E_i / \Delta t_i$. In one interval, this equation is not solvable (unless only one power state bit is active), but over time, an application generates a system of equations, one for each interval. When the system of equations is sufficiently constrained, a simple linear regression yields the individual power draws.

The regression works as follows. First, we collect the observed power states $\alpha_{i,j}$ in a matrix

$$A = \begin{bmatrix} \alpha_{1,1} & \cdots & \alpha_{1,n} \\ \vdots & \ddots & \vdots \\ \alpha_{m,1} & \cdots & \alpha_{m,n} \end{bmatrix},$$

filtering out columns in which the power state is always inactive (i.e., columns where $\alpha_{i,j}$ is always 0). Second, we determine the average aggregate power for each interval $y_i = \Delta E_i / t_i$. We collect the average aggregate power over all intervals in a column vector

$$\mathbf{b} = \begin{bmatrix} y_1 & \ldots & y_m \end{bmatrix}^T.$$

Third, we weight A and $\mathbf{b}$ as described in §2.5 of [5]. Finally, we estimate the unknown power draws $\mathbf{x}$ by solving the following convex minimization problem: $\underset{\mathbf{x}}{\text{minimize}} \; ||A\mathbf{x} - \mathbf{b}||_2$ subject to $\mathbf{x} \geq 0$.

2.2 Activity tracking

Quanto borrows from earlier work the concept of an *activity* as its resource principal. An activity is a set of operations whose resource consumption should be grouped together. We assign the energy consumption to activities that are defined by the programmer at a high level by following all operations related to an activity across hardware components on a single node and across the network. For a detailed overview of activity tracking in Quanto, see §3.1 of [5].

The mechanisms for tracking activities in Quanto are divided into three parts: an API that allows the programmer to create meaningful activity labels, a set of mechanisms to propagate these labels along with the operations that comprise the activity, and a mechanism to account for the resources used by the activities. For details about the activity tracking API and activity propagation, see §§3.2–3.3 of [5]. What is relevant is that in order to record the usage of resources for accounting and charging purposes, the operating system tracks activity changes of devices and logs these events to the serial port, and the rest of the accounting is done offline. For further details about the accounting of resources, see §3.4 of [5].

3 Motivation

The original Quanto prototype, as described above, relies on a log of two types of events: changes in power state for resources, and changes in activity for devices. Unfortunately, keeping such a log presents some scalability problems, especially in production sensor networks.

Logging large numbers of events to the serial port is unacceptable in production sensor networks not only because it wastes valuable time and energy, but also because doing so may have a noticeable impact on the operation of the system as a whole. In order to minimize these factors, three optimizations have been implemented in the Quanto prototype since its initial release. First, the standard TinyOS ActiveMessage headers were stripped down to the bare minimum, which saved considerable logging bandwidth. Second, the logging of events was relegated to a low-priority task, running only when the CPU was otherwise idle (but nevertheless always running to completion, even if a new activity was posted in the meantime). Third, the messages were compressed using Elias gamma coding [4] and the move-to-front (MTF) scheme of Bentley et al. [1]. Although this scheme saved considerable logging bandwidth, it incurred the additional overhead of compressing the messages on the node. Compression, like logging, was relegated to a low-priority task.

Unfortunately, these optimizations were not sufficient to allow Quanto to scale to intensive applications with very large numbers of power state changes and activity changes, such as the Collection Tree Protocol (CTP) [6]. To understand why, we examine the upper bounds on number of power state changes and number of activity changes per unit of time:

- The number of power state changes per unit of time depends on the application and is therefore unbounded. To make matters worse, more intensive applications are likely to generate more power state changes. The ramifications of this are that when smaller amounts of energy and processing time are available for logging, greater amounts of energy and processing time are required for logging. Hence this method does not scale.

- The number of activity changes per unit of time also depends on the application and is therefore unbounded. To make matters worse, Quanto statically assigns a fixed proxy activity to each interrupt handler and paints the CPU with the corresponding proxy activity every time an interrupt occurs. Since interrupts occur frequently, activity changes are also generated frequently. Hence this method does not scale, either.

These problems stem from the fact that all power state changes and activity changes are logged to the serial port and motivated us to develop a new logging module that accumulates this information on the node itself and reports it periodically.

Furthermore, recall that Quanto's offline processing tool groups all intervals from the log that have the same power state g (a particular setting of $\alpha_1, \ldots, \alpha_n$), adding the time and energy spent at that power state. Even if Quanto were to do this grouping on the node itself rather than offline, there can be up to 2^n such groupings, where n is the total number of power states over all resources in the system. In the Quanto prototype, where $n = 22$, this upper bound is very high. The fact that this upper bound usually exceeds the storage capacity of embeded devices motivated us to also develop a more efficient method of calculating the power draw of each component.

4 Design

In order to address the challenges described above, we developed both (a) a new logging module that accumulates and periodically reports the total time spent by each power state bit on behalf of each activity and (b) a more efficient method of estimating the power draw of each component. We describe these changes in detail below.

4.1 Cumulative logging module

Our logging module accumulates and periodically (in our prototype, every second) reports the total time spent by each power state bit on behalf of each activity, as well as the length of the interval and the aggregate amount of energy spent during the interval. For each power state bit, we (a) keep track of whether or not that bit is currently active; (b) maintain a list of activities on behalf of which that power state bit, if active, is currently working; and (c) maintain a timer that keeps track of how long that power state bit worked on behalf of its current set of activities. The main data structure in the module stores, for each reporting interval, the total time spent by each power state bit on behalf of each activity.

When a power state bit becomes active, we start the timer for that bit; when it becomes inactive, we stop its timer and charge the time that the bit was active to the activities on behalf of which it was working while it was active. Suppose there is a change in the set of activities on behalf of which a resource is working. For each power state bit corresponding to that resource, we update the set of activities for which that power state bit is working correspondingly. If the power state bit is currently active, we stop its timer, credit the time that it was active to the old set of activities, and reset its timer under the new set of activities. At the end of each reporting interval, we charge all active power state bits and reset their timers for the new reporting interval. Note that the set of activities in Quanto includes the unknown activity and the idle activity; therefore, as long as a power state bit is active, it must be doing work on behalf of at least one activity.

There are two policy decisions to make when doing the above: how should we account for time when a resource is working on behalf of multiple activities, and how should we aggregate information about activities originating at other nodes? In the first case, our logging module currently equally splits the time spent by each bit across all activities on behalf of which that bit was working. For example, if two activities are using the red LED, then each gets charged for half of the time. The second case is problematic because TinyOS does not feature dynamic memory allocation. We may not know how many other nodes exist in order to statically allocate a buffer; even if we did, we would not want to allocate a table of size an, where a is the total number of activities, and n is the number of nodes. Our logging module currently groups all activities from other nodes together, though other options are certainly possible. For example, we could aggregate activities from other nodes by activity, still keeping them distinct from activities that occur on the node in question, which has an upper bound of the total number of activities in terms of space. Another possibility is to aggregate activities from other nodes by node, which has an upper bound of the total number of nodes in terms of space. A final possibility is to aggregate activities from other nodes by both activity and node, which has an upper bound of the total number of nodes multiplied by the total number of activities in terms of space. Since the number of nodes is unbounded, the last two possibilities would be more difficult to implement for TinyOS.

Recall that when the activity associated with an interrupt or external event is not known, Quanto employs a *proxy activity* and later *binds* the proxy activity to the real activity associated with that interrupt or external event once the real activity becomes known. We account for the binding of a proxy activity to a real activity as follows. For each power state bit, we transfer the time that the bit spent on behalf of the proxy activity to the real activity. If we previously reported the time spent on behalf of the proxy activity, we can't do this transfer on the node itself, so we annotate the next report with an instruction that tells the offline processing tools, described below, to make an adjustment to the previous report.

4.2 Energy tracking

Suppose that the total number of intervals is m, and the total number of power state bits over all resources in the system is n. The input to the offline regression process in the original Quanto prototype is a log that records, for each interval i for which the power states are the same, the aggregate energy consumed during that interval (ΔE_i), the length of the interval (Δt_i), and the setting of each power state bit — active or inactive — during the interval ($\alpha_{i,1}, \ldots, \alpha_{i,n}$). In contrast, our cumulative logging module records, for each *reporting* interval i (in our prototype, every second), the total amount of time spent by each power state bit j during that interval ($t_{i,j}$), the length of the interval (Δt_i), and the aggregate amount of energy spent during the interval (ΔE_i). Based on this information, Quanto's offline processing tool generates one linear equation of the following form for each interval:

$$\Delta E_i = \Delta t_i \Delta E_i + \sum_{j=0}^{n} t_{i,j} p_{i,j},$$

where $p_{i,j}$ is the (unknown) power draw of the j-th power state bit during the interval and the first term of the addition is a constant representing an interval in which all power state bits are active. The average power over the interval $P_i = \Delta E_i / \Delta t_i$. In one interval, this equation is not solvable (unless only one power state bit is active), but over time, an application generates a system of equations, one for each interval. When the system of equations is sufficiently constrained, a simple linear regression yields the individual power draws.

The regression works as follows. First, we collect the observed times $t_{i,j}$ in a matrix

$$A = \begin{bmatrix} t_{1,1} & \cdots & t_{1,n} \\ \vdots & \ddots & \vdots \\ t_{m,1} & \cdots & t_{m,n} \end{bmatrix},$$

filtering out columns for which the power state bit was never active or always active (in which case those bits are grouped with the constant described above). Second, we collect the aggregate energy consumed during each reporting interval in a column vector

$$\mathbf{b} = \begin{bmatrix} \Delta E_1 & \ldots & \Delta E_m \end{bmatrix}^T.$$

Finally, we estimate the unknown power draws $\mathbf{x}$ by solving the following convex minimization problem: $\underset{\mathbf{x}}{\text{minimize}} \; ||A\mathbf{x} - \mathbf{b}||_2$ subject to $\mathbf{x} \geq 0$.

4.3 Activity tracking

In the original Quanto prototype, each node merely logged all activity changes to the serial port, and the rest of the processing was done offline. Our cumulative logging module reports the total time spent by each power state bit on behalf of each activity for each interval, so the only processing that remains to be done offline is to sum the total times for all intervals.

Reporting the total time spent by each power state bit on behalf of each activity for each interval has the additional advantage of providing greater observability. To expose this functionality to users, we developed a *top*-like tool that summarizes the time spent per activity per power state bit for each reporting interval.

5 Implementation details

Recall that to be acceptable for use in production sensor networks, profile collection must have a negligible impact on the operation of the system as a whole. Implementing such a system is challenging due to the limited processing power and storage available on embedded devices, with clock speeds on the order of tens of megahertz and storage capacities on the order of kilobytes of RAM. Our implementation consists of three components: First, we implement a mechanism for handling deferred work from interrupts in TinyOS and utilize it to perform our accounting with minimal impact to the rest of the system. Second, we compress the data being sent at each interval in order to save bandwidth on the serial port during logging. Finally, we log each report to the serial port. We describe this process in detail below.

5.1 Accounting

Recall that we must account for time whenever there is a change in the set of activities on behalf of which a particular resource is working. Due to Quanto's use of proxy activities, such a change occurs for the CPU with every interrupt. Unfortunately, doing this accounting at every interrupt is prohibitively expensive. For each of the CPU's power state bits, we must split the time spent by the bit across all activities on behalf of which it was working. When the proxy activity is bound at a later time, we must also transfer the time that each power state bit spent on behalf of the proxy activity to the real activity. These calculations require too many CPU cycles to be done in the context of an interrupt handler, which must return quickly in order for the system as a whole to make progress.

To deal with this problem, we implemented a mechanism for handling deferred work from interrupts in TinyOS. Whenever an activity or power state change takes place (whether in the context of an interrupt or not), we log the change to a deferred-work queue and quickly return. A separate TinyOS task runs periodically to process events from the deferred-work queue. Normally, this task is low-priority; that is, it runs only when the CPU would be otherwise idle. However, if the number of unprocessed events in the deferred-work queue crosses a certain threshold (in our prototype, half the size of the queue), this task is run at a higher priority in order to prevent the queue from getting full and events from being dropped.

5.2 Compression

In order to save bandwidth on the serial port during logging, we compress all outgoing reports with a low-priority task that runs only if the CPU is otherwise idle. Recall that the main data structure in each report stores, for each reporting interval, the total time spent by each power state bit on behalf of each activity. This table can be very sparse for two reasons. First, in many applications a large number of power state bits are never active. Second, there are many activities that are only active for a subset of power state bits (for example, proxy activities only ever do work on behalf of the CPU's power state bits).

The sparse nature of this table motivated us to develop a compression scheme based on run-length encoding (RLE). We first developed a bitwise scheme that employs Elias gamma coding [4]. In our bitwise compression scheme, the first bit is the same as the first bit of the uncompressed table in order to indicate the initial polarity for decoding. Following the first bit are a series of Elias gamma coded run lengths of successive bit runs in the uncompressed table, alternating polarities.

Although this scheme yielded great space savings, the number of cycles needed to perform the compression on the node was too high for use in production sensor networks. As a result, we settled on a less sophisticated bytewise compression scheme without the use of Elias gamma coding. Our bytewise scheme consists of a series of two-byte pairs, the first of which indicates the run length and the second of which is the data byte.

Figure 1 shows a comparison between the types of compression in one of our tests (a 48-second run of an application that transmits data over the wireless radio). Without compression, Quanto sent seven times as many messages over the serial port and was so inefficient that

	Uncompressed	Elias gamma RLE	Simple RLE
Total messages	1,106	153	155
Total number of bytes (encoded)	-	4,399	5,796
Total number of bytes (decoded)	159,580	52,071	52,080
Compression factor	-	11.84	8.98
Time spent compressing (sec)	-	5.07	0.32
Time spent writing to serial port (sec)	N/A	0.86	1.04
Overhead in terms of total time	N/A	11.40%	2.58%

Figure 1: Comparison of run-length encoding (RLE) compression schemes. The Elias gamma coded RLE scheme yielded a high compression factor, but its overhead in processing time was unacceptably high compared to that of the simpler RLE scheme.

it was impossible to obtain reliable figures regarding the time spent writing to the serial port and the overhead in terms of total time. The Elias gamma coded RLE scheme achieved a high compression factor (11.84) at the cost of a large amount of time spent compressing (an overhead of 11.40% of the total time). The simple RLE scheme achieved a respectable compression factor (8.98) but spent much less time compressing (an overhead of only 2.58% of the total time).

5.3 Reporting

Once the outgoing report has been compressed, our prototype logs it to the serial port. Logging, like accounting and compression, is implemented as a low-priority task in order to minimize Quanto's impact on the system as a whole. We employ a double buffer so that accounting for the next interval can begin even while the report for the current interval is being compressed and sent out.

6 Evaluation

To test our implementation, we instrumented two simple applications provided with TinyOS: *Blink* and *RadioCountToLeds*. Our test environment consisted of two nodes based on the Epic platform [3], which incorporates iCount into a custom sensornet node. This platform uses the Texas Instruments 16-bit MSP430 microcontroller with 48 KB of internal flash memory and 10 KB of RAM and an 802.15.4-compliant CC2420 radio. The platform also includes three LEDs. Each node was connected to a Digi external serial server with a RealPort COM port redirector. We ran each of the two applications for 48 seconds.

Power state bit	P_{avg} (mW)	I_{avg} (mA)
LED0	7.54	2.283
LED1	6.06	1.837
LED2	2.44	0.740
MSP430 Active	11.71	3.549
MSP430 LPM1	1.08	0.328
Const.	1.25	0.377

(a) Node A

Power state bit	P_{avg} (mW)	I_{avg} (mA)
LED0	7.25	2.197
LED1	6.87	2.083
LED2	1.91	0.579
MSP430 Active	8.85	2.683
MSP430 LPM1	1.28	0.387
Const.	1.36	0.411

(b) Node B

Figure 2: Breakdown of energy usage by hardware component during a 48-second run of *Blink*

6.1 *Blink*

Blink is the simple *hello world* application in TinyOS. It starts three independent timers with intervals of 1, 2, and 4 seconds. When these timers fire, the red, green, and blue LEDs are toggled, such that in eight seconds Blink goes through eight steady states, with all combinations of the three LEDs on and off.

We ran the regression as described in §4.2 in order to estimate the power draw of each component. Figure 2 shows the result in current and power. As expected, the first LED draws the most power, followed closely by the second LED, followed distantly by the third LED. As expected, the CPU draws a large amount of power when it is active and a very small amount of power when it is in low-power mode (LPM). Furthermore, these results match the results obtained using the original Quanto prototype.

Figure 3 shows the amount of time spent by each power state bit on behalf of each activity. As expected, each of the LED power state bits spends almost all of its time on its corresponding activity. This figure also shows that the nodes' CPUs spent about 90% of the total time of the run in low-power mode (LPM) and were only active for about 10% of the total time. Of the time the CPU was active, a significant portion was spent accounting (the *Log* activity), compressing the data (the *Compress* activity), and sending out the reports to the serial port (the *Writer* activity). Altogether, the instrumentation overhead added by Quanto in terms of CPU time is about 5% of the total time, all of which was spent when the CPU would have otherwise been idle.

6.2 *RadioCountToLeds*

RadioCountToLeds is a simple application in TinyOS to test mote-to-mote radio communication and timers. A sender maintains a 4Hz counter, broadcasting its value in an ActiveMessage packet every time it gets updated. A receiver node that hears a counter broadcast displays the bottom three bits on its LEDs.

As before, we ran the regression as described in §4.2. Figure 4 shows the result in current and power. As expected, the sender draws the most power when transmitting packets over the

Activity	Time(s)	% active	% total
MSP430 Active			
Log	1.89	37.67	3.62
Writer	0.92	18.29	1.76
Other Node	0.84	16.63	1.60
Timer	0.59	11.65	1.12
Red	0.29	5.84	0.56
Compress	0.17	3.39	0.33
Green	0.15	2.97	0.28
Idle	0.10	2.08	0.20
Blue	0.07	1.46	0.14
MSP430 LPM1			
Idle	47.02	99.32	89.78
Timer	0.31	0.65	0.59
LED0			
Red	26.15	99.81	49.94
LED1			
Green	25.99	99.91	49.62
LED2			
Blue	26.15	99.95	49.93

(a) Node A

Activity	Time(s)	% active	% total
MSP430 Active			
Log	1.91	37.79	3.63
Writer	0.92	18.27	1.75
Other Node	0.83	16.50	1.58
Timer	0.59	11.71	1.12
Red	0.30	5.85	0.56
Compress	0.17	3.38	0.32
Green	0.15	2.93	0.28
Idle	0.11	2.09	0.20
Blue	0.07	1.46	0.14
MSP430 LPM1			
Idle	47.17	99.32	89.79
Timer	0.31	0.66	0.59
LED0			
Red	26.12	99.81	49.72
LED1			
Green	26.15	99.91	49.76
LED2			
Blue	26.35	99.95	50.15

(b) Node B

Figure 3: Amount of time spent by each power state bit on behalf of each resource during a 48-second run of *Blink*

Power state bit	P_{avg} (mW)	I_{avg} (mA)
CC2420 Listen	0.00	0.000
CC2420 Tx 31	458.37	138.901
CC2420 Tx FIFO	79.80	24.182
MSP430 Active	42.57	12.899
MSP430 LPM1	0.00	0.000
Const.	0.00	0.000

(a) Sender

Power state bit	P_{avg} (mW)	I_{avg} (mA)
CC2420 Listen	0.00	0.000
CC2420 Rx	0.00	0.000
CC2420 Rx FIFO	133.70	40.514
LED0	6.50	1.969
LED1	6.47	1.961
LED2	1.57	0.475
MSP430 Active	13.81	4.184
MSP430 LPM1	1.30	0.394
Const.	5.40	1.637

(b) Receiver

Figure 4: Breakdown of energy usage by hardware component during a 48-second run of *RadioCountToLeds*

radio and the receiver draws the most power listening for packets on the radio. The receiver also draws power in order to blink the LEDs when it receives a message; the draw for each LED is in the expected proportions. On both the sender and the receiver, the CPU draws more power when it is active than when it is in low power mode (LPM).

Figure 5 shows the amount of time spent by each power state bit on behalf of each activity. As expected, the LED power state bits on the receiver spend all of their time on behalf of the sender. The sender was idle in low-power mode (LPM) for about 50% of the total time, while the receiver was idle for about 60% of the total time. The majority of active CPU time on each node was spent working on behalf of the other node. Altogether, the instrumentation overhead added by Quanto in terms of CPU time is about 10% of the total time on the sender and about 4% of the total time on the receiver, all of which was spent when the CPU would have otherwise been idle.

7 Future work

Our improvements have greatly increased Quanto's scalability for production sensor networks, but they rely on individual nodes having enough CPU time to do accounting, compression, and logging at regular time intervals. Sudden bursts in CPU activity can interfere with the processing of events in our logger's deferred-work queue. When this queue becomes full, power state change events and activity changes may be dropped, leading to inaccurate reports. We currently use a crude mechanism to prevent this from taking place. Normally, the deferred-work queue is processed with low priority; that is, it is processed only when the CPU would be otherwise idle. However, if the number of unprocessed events in the deferred-work queue crosses a certain threshold (in our prototype, half the size of the queue), the queue is processed with high priority. Though few, if any, events are dropped with this method, it

Activity	Time(s)	% active	% total
MSP430 Active			
Other Node	8.83	37.14	16.69
Proxy Port 1	7.71	32.41	14.56
Log	4.11	17.28	7.77
Timer	1.21	5.10	2.29
Writer	1.04	4.39	1.97
Count to LEDs	0.35	1.49	0.67
Compress	0.32	1.36	0.61
Idle	0.19	0.78	0.35
MSP430 LPM1			
Idle	27.59	98.13	52.13
Timer	0.33	1.18	0.63
Proxy Port 1	0.14	0.51	0.27
Log	0.05	0.17	0.09
CC2420 Listen			
Idle	2.19	81.24	4.13
Proxy Port 1	0.46	17.26	0.88
Unknown	0.04	1.33	0.07
CC2420 Tx 31			
Proxy Port 1	1.18	82.93	2.23
Idle	0.23	16.33	0.44
CC2420 Tx FIFO			
Proxy Port 1	0.40	99.06	0.75

(a) Sender

Activity	Time(s)	% active	% total
MSP430 Active			
Other Node	8.65	42.19	16.20
Timer	3.85	18.81	7.22
Proxy CC2420 Rx	3.25	15.85	6.08
Proxy UART0 Rx	1.49	7.25	2.78
Proxy Port 1	1.41	6.86	2.63
Writer	1.11	5.42	2.08
Compress	0.32	1.55	0.60
Log	0.26	1.25	0.48
Idle	0.13	0.63	0.24
Count to LEDs	0.02	0.10	0.04
MSP430 LPM1			
Idle	32.54	98.99	60.98
Timer	0.33	1.01	0.62
CC2420 Rx			
Proxy CC2420 Rx	0.19	100.00	0.36
CC2420 Listen			
Idle	2.68	97.96	5.02
Unknown	0.04	1.36	0.07
Proxy CC2420 Rx	0.02	0.68	0.03
CC2420 Rx FIFO			
Proxy CC2420 Rx	2.52	96.08	4.71
Idle	0.10	3.92	0.19
LED0			
Other Node	27.45	100.00	51.45
LED1			
Other Node	22.82	100.00	42.77
LED2			
Other Node	25.78	100.00	48.31

(b) Receiver

Figure 5: Amount of time spent by each power state bit on behalf of each resource during a 48-second run of *RadioCountToLeds*

can still be problematic. By the conventions of TinyOS, all tasks must run to completion. Even if a high-priority processing task is scheduled, the operating system may be busy with another high-priority task for a long period of time before running the processing task. Furthermore, running the accounting task with high priority while leaving the compression and reporting tasks at low priority can starve out the compression and reporting tasks, leading to inaccurate reports. It would be useful to implement more sophisticated scheduling and deferred processing mechanisms from conventional operating systems. For example, the queue could be processed incrementally by a low-priority interrupt handler with a fixed upper bound on time. Similarly, a hierarchy for low-priority tasks could be introduced to prevent compression and reporting from being starved out by accounting.

With our improvements in place, we hope to see Quanto used to profile more production sensor networks. Our preliminary tests with the Collection Tree Protocol (CTP) [6] show that our new methods for online activity tracking and estimating energy scale much better than the initial Quanto prototype. With the scheduling changes described above, we hope that a full time and energy profile of CTP in a production sensor network is within reach. From there, the accounting optimizations described in §4.1 can yield even further visibility into network behavior.

Our improvements also bring Quanto closer to a point where we can do all energy tracking on the node itself without the use of an offline regression tool. In our mechanism for determining the energy breakdown by component, the number of equations that must be solved is the same as the number of intervals for which data was collected. If a node is particularly idle, this time could be used to perform the regression on the node itself, and the results of a completed regression could be used to inform the next regression. Once a regression has been completed, exposing the results to applications for use in, for example, decisions about when to turn off the wireless radio, would be merely a matter of semantics.

8 Conclusion

Profiling time and energy usage in production sensor networks is an area still in its infancy. Developing an accurate profile in production sensor networks necessitates pushing the hardware to its limits and carefully tuning the operating system's scheduler, the logger's compression scheme, and the profiling algorithm itself in order to strike the ideal balance between the consumption processing time and I/O bandwidth in order to minimize the impact of profiling on the operation of the system as a whole. Our improvements to Quanto have made it a more viable choice for performance analysis in production sensor networks, where it provides much-needed visibility into network and device behavior.

9 Acknowledgments

We would like to thank Prof. Rodrigo Fonseca and Marcelo Martins for their valuable assistance throughout the duration of this project.

References

[1] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29:320–330, April 1986.

[2] Prabal Dutta, Mark Feldmeier, Joseph Paradiso, and David Culler. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks*, IPSN '08, pages 283–294, Washington, DC, April 2008. IEEE Computer Society.

[3] Prabal Dutta, Jay Taneja, Jaein Jeong, Xiaofan Jiang, and David Culler. A building block approach to sensornet systems. In *Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems*, SenSys '08, pages 267–280, New York, NY, November 2008. ACM.

[4] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, March 1975.

[5] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: Tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '08, pages 323–338, Berkeley, CA, December 2008. USENIX Association.

[6] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection Tree Protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 1–14, New York, NY, November 2009. ACM.

[7] Marcelo Martins, Rodrigo Fonseca, Thomas Schmid, and Prabal Dutta. Network-wide energy profiling of CTP. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 439–440, New York, NY, November 2010. ACM.

[8] Gilman Tolle and David Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proceedings of the 2nd European Workshop on Wireless Sensor Networks*, EWSN '05, pages 121–132, February 2005.