Lightweight Deployable Software Monitoring for Sensor Networks

Mustafa Hammad
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003
mhammad@cs.nmsu.edu

Jonathan Cook
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003
joncook@nmsu.edu

Abstract—Most efforts at analyzing the behavior and performance of sensor network software have been simulation-based, but deployment in the field can bring about conditions that were unexpected, or just too hard to simulate. Thus monitoring the software in the field would be a valuable capability for engineering of sensor networks. We have developed a lightweight instrumentation tool for TinyOS, called LIMOW, that is deployable onto sensor hardware, is low cost in overhead, and is configurable and controllable in terms of how much it monitors and how much overhead is tolerable. We show both behavioral and timing results from our tool, and compare results with other monitoring approaches.

I. INTRODUCTION

Wireless sensors are very resource-constrained devices, yet software development for these devices is certainly just as challenging for them as for other common computational platforms. This is because the software, though functionally perhaps more simple than a large desktop application, runs in an asynchronous event-driven environment, inherently performs communication, and must ultimately deal with coordination issues among the whole sensor network. It essentially comprises a large parallel, distributed computational system.

When the software is running on the actual hardware, it is almost completely opaque due to the limited platform and the few I/O channels that exist to communicate with the device. Thus simulators have been the accepted method for testing, debugging, and analyzing the behavior of the software that will be deployed into the field. Since desktop development computers are much more powerful than the sensor devices, simulating them is not prohibitively slow and an extremely large amount of information can be gleaned about the software.

Simulators, however, have drawbacks. One, the simulator may not exhibit complete fidelity to the real world. Thus some possible environmental occurrences may not be tested before deployment. Two, the developer must craft input for the simulator, typically creating some form of model of their expected environment. This can be tedious and error-prone. Finally, developers rarely attempt to create simulations that approach the real complexities of the deployed environment; rather, they make their simulation “just complex enough” to give them some level of confidence that their software will behave correctly in deployment.

So while simulators are and will be very useful tools for sensor development, in-the-field software monitoring can provide added capabilities that help the developer better understand the system they are creating or deploying, and ultimately help them improve it and ensure that it is operating as intended.

We have developed a lightweight monitoring tool for wireless sensor software, called LIMOW. LIMOW instruments all or some of the sensor software, locally collects data, and then at specified times sends data back to a gateway for permanent collection and inspection. It operates on programs written in nesC within the TinyOS framework [1]. This paper presents the design and implementation of LIMOW, its benefits, and an evaluation of its efficacy.

II. BACKGROUND

We previously applied a lightweight monitoring technique [2] to sensor software being executed under the TOSSIM simulator [3]. Since TOSSIM compiles to native executables, internal inspection of the program behavior is still quite opaque, and our monitoring enabled the developer to see the internal behavior and better estimate timing measurements.

The TinyOS toolchain uses a preprocessor to convert the nesC program into a C program, then uses a C compiler to compile that program to a native executable. The basic idea in our previous work was to use the function call hook that is inserted automatically by a compiler when the profiling compilation option is selected. We overrode the normal profiling function with our own, so as to perform our own monitoring rather than profiling. By also attaching an end-of-function hook, we could capture and record the occurrence and time of each function call and return.

This data is then mapped from C functions back to nesC concepts, and can be used to produce diagrams, summary results, or whatever the developer’s need is. A key idea here is that the monitoring is done at a C-function level, but the developer is thinking about nesC concepts: tasks, events, commands, etc.; a complete toolset must present data in the concepts the developer is using.

This work was supported by the National Science Foundation under grants CCF-0541075 and CRI-0551734. Views expressed herein are those of the authors and do not necessarily represent those of the Foundation.
III. METHODS

Initially our desire was to simply use the capabilities we developed for instrumenting TOSSIM executables directly on the hardware executables. There were a couple of setbacks to this, however.

One, in the TinyOS toolset, the gcc compiler for our sensor platform (mica2) takes the profiling flag but does not instrument the code with a call to any profiling function. Instead, it inserts two no-operation instructions into each function at the point where the profiling call is typically inserted. The reason for this is probably the deliberate disabling of profiling because it is not implemented in the TinyOS code that gets statically linked to the nesC application.

Two, we needed a mechanism to control the amount of instrumentation added, rather than allowing the compiler profiling flag to simply blindly instrument all functions. On real hardware with real performance issues, we need to “dial in” the correct amount of instrumentation. Since TinyOS is compiled with the nesC application, the full instrumentation that the compiler would do with a profiling flag would be cost-prohibitive on the real hardware.

Overall, our goal was still to add instrumentation without any modification to the source code.

A. Framework

The approach that solved these issues, related to what worked for TOSSIM, is shown in Figure 1. For both compilation paths, the TinyOS toolset translates nesC into C code, then uses gcc to compile the programs. Rather than using the profiling flag, we use the ability of gcc to produce assembly code for the program it is compiling. Assembly code produced by a compiler is very regular and easy to parse and edit automatically, which makes it an ideal medium for instrumentation. Source code instrumentation generally requires a full language parser and can be fraught with syntactic pitfalls, while binary editing must deal with patching up branch/jump addresses and other machine code issues.

Thus, with the assembly code, we take as input a list of function names and process the assembly code of the program, adding instrumentation to only the functions that were specified. This modified assembly code is then fed back to the normal TinyOS compilation process, and a deployable executable is obtained. The only drawback we have experienced with this approach is that we must turn off compiler inlining of function calls; otherwise some parts of the program “disappear” at the assembly level. Our evaluation section addresses this issue in more detail.

B. Monitoring Component

Since our monitoring instrumentation now needs to interact with TinyOS and perform radio communication, we followed the nesC component model of development and created a Monitor nesC component. The instrumentation that is inserted automatically invokes a Monitor::AddToLog command to record the proper information in the local monitor log. This means that currently, one does need to wire the Monitor component to the application, but this can be automated.

Eventually, the monitor on a sensor must send its local data back out of the network and to a gateway device that will relay it to the proper location. Our monitor has two ways of doing this. The first is simply for the monitor to send data at some specified time interval. The monitor automatically sets up a timer for sending the log, and this timer is user-configurable. The monitor also has a configurable gateway ID specifier, and this does not need to be the same as the application gateway (but could be). Separate gateways make it easier to keep the monitoring data isolated, but a single gateway could filter out the monitoring data and process it separately from the application data.

The second is for the application to directly invoke the Monitor::SendLog command to send its current data and clear its local data log. While this violates the “no changes” rule for instrumentation, it is very efficient because the developer knows when the application is finished with a round of processing and is about to go back to sleep, or when an error condition may have occurred that would be useful to report data on. One of our future goals is to automatically discover these points in the program.

C. Managing the Local Log

Since sensors are very resource-constrained, it is important to control the amount of resources, including memory, that the monitor uses. We do this by not allowing the monitor to have an unbounded local log buffer; rather we fix the amount of

\[\text{Monitor::SendLog}\]

1Our earlier TOSSIM-based approach did indeed capture full information about the internal behavior of TinyOS, which is useful in its own right for those who are working on TinyOS itself.
memory it might use, and use the area as a circular queue for log data. This means that if the monitor does not send the log within the time it takes to fill it, it will start losing data as the monitor overwrites older logged data. This queue can be a two-level queue, where the second level uses flash storage, and only a current block is updated in RAM.

D. Timing

When we first deployed our monitoring onto real hardware, we immediately noticed that the timing information we could get was very coarse grained and limited. The system “tick” rate seemed to be very low; this we found, was set in TinyOS itself, even though higher rates were supported by the hardware. Once tracked down, we found it fairly simple to adjust the internal TinyOS tick rate. TinyOS selected a tick rate of 1/1024 second; we changed this to the maximum indicated in TinyOS code, 1/32768 second. The tick rate effected some scale factors elsewhere in the TinyOS code, so care must be taken when modifying it. This change gave us a 30-microsecond accuracy, as opposed to 1-millisecond accuracy with the default tick rate.

The second issue with timing was the default 8-bit value that was obtainable from Counter::getCounter. This would easily wrap around and would make it harder to correctly determine elapsed times. We again eventually found a convenient place in TinyOS to implement a larger counter, ultimately selecting 32 bits for a tick counter.

Internally, a Clock component responds to the hardware tick interrupts, and fires an event that is received by the Timer component. The Timer component manages the requested timer intervals of components connected to it, and only fires Timer events when some component needs one. We added a 32-bit tick counter to the Timer component, and a command readCounter; this command returns the tick counter plus the most recent Clock.getCounter value. The 32-bit counter will wrap around over about 36 hours, but we expect that local data will be delivered well within this time frame. We do not deal with cross-node time correlation; many techniques have already been developed to support this.

E. Diagramming Software Behavior

Once the data is collected at the base station, it can be used to generate useful information for the developer. There are myriad of uses for the data we collect, but here we present two, a detailed view of timed program behavior, and a summary view of timed program behavior. These views are generated automatically by programs that are part of the LIMOW toolset. Such views are limited to those parts of a program that are chosen to be instrumented, and thus typically do not show everything the program is doing; in particular, neither the internal TinyOS behavior is typically instrumented (though it could be), nor is the monitor itself and other parts of the application deemed uninteresting for the issue at hand.

Figure 2 shows a detail view for a simple example program, the canonical BlinkTask program, which uses a task to blink the LEDs. This diagram shows just one occurrence of the behavior; the actual data logged has repeated instances of this behavior for as long as the program was allowed to run. For this program, we only instrumented the timer event, the main task, and the command to toggle the LED. Note that each is denoted by its respective mode of execution: fire for the event, post and run for the task, and call for the command.

Figure 3 shows the summary view for the same BlinkTask program. For a very repetitive program, the entire behavior is not useful unless one is trying to spot an anomaly, but a summary of the behavioral characteristics is useful. This figure provides those summary statistics for the timer event and the task. Again, we provide specific information based on the nesC concepts, not just the raw C functions that underlie them. For the event, the average, minimum, and maximum interval times are displayed, and then the average, minimum, and maximum execution times. For the task, the average, minimum, and maximum delay time from post to run is displayed, along with the execution times. Note that this program asks for a timer rate of 1000 “milliseconds”, but in TinyOS the timer rate is in binary milliseconds, or 1/1024 seconds, so that the average event interval shown of 976.6 milliseconds is indeed accurate.

Figure 4 shows a behavioral snippet from a program that reads a light sensor and sends the data as a message, and receives data messages from other sensors, displaying their values on its own LEDs. The figure shows data logged some
Fig. 4. Detail diagram of Sense&Send application.

time after the program began; it shows a dataReady event, an execution of an already-posted sendData task, a message reception including setting the LEDs, a local timer firing that starts a light sensor reading, and finishes with data sending.

This example shows what can still be displayed even if some of the local monitoring log has been lost (due to overwriting old data before it was sent). The first Fire on the diagram is marked with an asterisk because it was not in the log but rather inferred by our visualization tool, since the log contained a return without an accompanying fire. The log information tells the visualization tool exactly how much data was lost, so it can report that 12 actions are missing from what was prior (not shown in this figure). Note that the tool does not infer the complete action within the event handler, because at the end of the figure it is seen that the dataReady event posts the sendData task, but this is not shown at the top of the figure.

While the detail diagrams are useful for node-level behavior but not useful for network-level behavior (since we do not yet do time correlation), the summary diagrams would be useful for some network-level behavior. In particular, the timing statistics would be useful to compare across nodes in a network, identifying potential bottleneck or overutilized nodes, as well as underutilized ones.

IV. Evaluation

To demonstrate the utility of LIMOW, we performed several evaluations. Of concern were whether it captured and report accurate timing information, and what the overhead in terms of power consumption and memory usage was. In order to evaluate these we used the Avrora simulator [4]. An interesting note is that we are using a simulator to validate our expected hardware behavior, and in some cases the simulator itself may not be validated. In these cases we actually are helping to validate the Avrora simulator itself, insofar as evidence of correlative agreement between two tools (along with first principles reasoning on their respective correctness) helps to establish the fidelity of both.

A. Time

Our first need is to measure the overhead in time that our instrumentation uses. Unfortunately, on real hardware we cannot measure an un-instrumented version. Therefore, we measure the time for both the instrumented version and the uninstrumented version in Avrora, and from this evaluate the overhead. At the same time, we obtain real measurements from the hardware for the instrumented version, and compare this to Avrora output. This is, as far as we can tell, the first “large-scale” validation of Avrora output, since Avrora developers had the same problem; they validated some small test cases, but essentially assumed correctness beyond that [4].

Table I shows measurements of the duration of a task that toggles the LEDs from once to 1000 times; this varies the length of the task from very short to extremely long; a typical nesC task might be on the order of the 10-LED-toggle task. Each value shown is an average of five measurements.

Our monitoring measurements of time are shown in the “time 3” column. Time columns are marked with numerical identifiers and then referred to in the difference columns by these numbers. The difference 1 – 2 shows the difference between the instrumented and non-instrumented versions of the program in Avrora, where the instrumentation is only on

We report Avrora times based on 135.6 nanoseconds per cycle, which more accurately corresponds to the 7.3728MHz clock than does the 136 nanoseconds used in the Avrora reporting.

2
TABLE I
TASK DURATION MEASUREMENTS USING AVRORA AND HARDWARE MONITORING (SEC).

<table>
<thead>
<tr>
<th>LEDs toggled</th>
<th>Avrora Instrumented ticks</th>
<th>Avrora Non-Instr. ticks</th>
<th>H/W Monitor ticks</th>
<th>Differences</th>
<th>Hardware w/ inline ticks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>time 1</td>
<td>time 2</td>
<td>time 3</td>
<td>1 - 2</td>
</tr>
<tr>
<td>1</td>
<td>587</td>
<td>7.96E-05</td>
<td>241</td>
<td>4.27E-05</td>
<td>4.7E-05</td>
</tr>
<tr>
<td>10</td>
<td>2761</td>
<td>3.74E-04</td>
<td>2415</td>
<td>3.85E-04</td>
<td>4.7E-05</td>
</tr>
<tr>
<td>100</td>
<td>24496</td>
<td>3.32E-03</td>
<td>24150</td>
<td>3.49E-03</td>
<td>4.7E-05</td>
</tr>
<tr>
<td>1000</td>
<td>242613</td>
<td>3.29E-02</td>
<td>242034</td>
<td>3.49E-02</td>
<td>7.9E-05</td>
</tr>
</tbody>
</table>

the toggle task. For 1 LED the difference is large, with the instrumented version more than twice as slow as the non-instrumented version. As the computation of the application increases, however, the overhead of instrumentation drops significantly since the task body in between the instrumentation sites gets longer. For the 10-LED version the overhead is about 10%, and gets vanishingly small for the 100-LED version. The overhead is consistent between the 1, 10, and 100-LED versions, indicating the same function call overhead for the instrumentation, regardless of the duration of the function; we believe the 1000-LED version varies from this because it is a long-enough task to incur the cost of rolling over the native TinyOS 8-bit tick counter into the 32-bit counter.

The difference 1 − 3 in the second table compares the actual hardware-monitored time with the Avrora time. The 1-LED version accumulates too much noise to be of value, since the number of ticks per invocation is so low. However, even for the larger computations, the difference between Avrora and the real hardware is still existent. For the 100 LED version, taking the hardware ticks and assuming a 7.3728MHz clock, the hardware time is reporting 25740 cycles, versus the 24496 reported by Avrora; these are within about 95% of each other, but a 5% difference is non-trivial for a cycle-based simulator. We do not yet know the source of the discrepancy; perhaps it is as simple as the real hardware clock tolerances. We are conducting more experiments to evaluate this, but from these initial results we think our numbers show larger-scale verification of Avrora than has previously been seen.

Finally, the last three columns in the second table represent the software performance on the hardware when compiled with function inlining enabled. In this case we hand-instrumented the source code with calls to our monitor code, and then compiled it from source. As can be seen, the inlined version is significantly more efficient and faster than the non-inlined version, which is almost four times slower. This extreme difference is because of the simple regularity of our LED-toggling task, and though the inlined versions should typically have somewhat better performance, we do not think this result will be the normal case. Others have seen around 30% slowdown on common applications without inlining [5]. Our future work includes investigating selective disabling of inlining so that instrumentation points are available but the rest of the program can be efficiently compiled.

B. Power

We used Avrora to obtain initial estimates of power consumption overhead caused by monitoring. Table II shows some basic comparisons between the normal inlining compilation mode, not inlining, and adding in the monitor (also without inlining). The results for the BlinkTask program are unrealistic because the basic program does not use the radio at all, while the monitor does. This obviously adds considerable overhead in terms of power consumption, in our experiment incurring 400% of the non-monitored program’s power consumption.

A second, more realistic program, is one that activates a local sensor, reads it, and then sends the data on through its radio. This program is also evaluated and shown in Table II. Here, the monitoring incurs a much more reasonable 3% overhead as compared to the program without monitoring, and removing the inlining during compilation incurs about 1% of that overhead.

The actual overhead experienced by a monitored application in the field would be highly dependent on how much monitoring was deployed, how big the local data log was made, how often the monitor log data was sent out, and how many hops the log data must make to get to a gateway. Since these are all configurable and/or dependent on a particular deployment, our results here are only meant to show that monitoring can be of reasonable cost.

We also did an analysis of how much power cost the increased hardware tick rate entails. Over a variety of simulations using Avrora, we consistently found an added power consumption of 8.95e-05 J/sec, which is less than a 1% increase in the idle power consumption of the standard tick rate, and is extremely insignificant with an actual program running and using the radio.

C. Memory Usage

Table III shows the memory usage over three versions of applications: one with normal compilation allowing inlining, one with compilation not allowing inlining, and one with the monitor attached and also not allowing inlining. This is shown both for the BlinkTask program and for the Sense&Send program. The final column shows the percent of ROM used
TABLE III
MEMORY USAGE ANALYSIS.

<table>
<thead>
<tr>
<th>Application</th>
<th>Bytes used</th>
<th>ROM</th>
<th>RAM</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlinkTask, inline</td>
<td>1584</td>
<td>167</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>BlinkTask, no inline</td>
<td>2878</td>
<td>103</td>
<td>182</td>
<td></td>
</tr>
<tr>
<td>BlinkTask, w/ monitor</td>
<td>15700</td>
<td>1202</td>
<td>991</td>
<td></td>
</tr>
<tr>
<td>Sense&amp;Send, inline</td>
<td>12146</td>
<td>623</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Sense&amp;Send, no inline</td>
<td>15906</td>
<td>623</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td>Sense&amp;Send, w/ monitor</td>
<td>16774</td>
<td>1431</td>
<td>138</td>
<td></td>
</tr>
</tbody>
</table>

relative to the no-monitor, inlining version. Since in TinyOS many routines are called only in one place, inlining can actually reduce code size. The two programs roughly incur an 80% and a 30% space overhead for not inlining. Others have seen similar results [5]. The monitored BlinkTask program literally explodes in size, but this is because the bare program does not link in any radio communication components, while the monitored version does. The Sense&Send program is more realistic, where the monitored program incurs about 7% more overhead above the no-inlining cost.

V. RELATED WORK

There is quite a bit of work on visualizing wireless sensor software behavior, and in this work various instrumentation approaches have been taken. In this section we compare and contrast LIMOW with other significant work.

Dalton et al. [6], [7] have developed a testbed for visualizing the behavior of TinyOS applications. Like our work, they instrument a call to a logging component at the entry and exit of selected actions, and the final output is displayed for the user as UML sequence diagrams. Their entire framework has more support for front-end activity selection and back-end visualization tasks than ours does, but on the instrumentation part they perform source-level instrumentation, purposely do not collect timing information on actions, and do not use radio communication to transmit the local event log but rather require a new binary image to retrieve the log. Our instrumentation methods could add new and different capabilities to their framework.

Cao and Abdelzaher [8] include an event trace logger in their LiteOS operating system to extract the application behavior if needed. The logger can trace the sequence of both kernel and application events, but does not record any timing information. It also does not actively send this information through the radio but allows it to be accessed using the Unix-like commands that LiteOS implements.

Woehrle et al. [9] created EvAnT, a framework to analyze behavior as described in event trace logs. Their framework supports queries and event abstractions, but they do not implement their own event collection mechanisms. Our LIMOW instrumentation framework could be used to produce event traces to use in their analysis framework.

Krunic et al. [10] developed NodeMD, a deployment management system which detects runtime failures and reports data back to a basestation that can evaluate the failure cause. It implements a pre-defined set of failure checks, such as stack overflow and deadlock/livelock, and also send back a log of recent activity over a pre-defined set of events. Its focus is on highly optimized failure checking and data storage. NodeMD also has an API to allow an application developer to assert their own failure condition checks.

VI. CONCLUSION

This paper presented a lightweight in-the-field monitoring framework for wireless sensor networks using the TinyOS platform. Our work still has significant directions for the future. One direction will be to intelligently use a limited data buffer for logging events rather than just a circular queue, with the goal of having "graceful degradation" in the degree of information that is kept about the application progress. We need to better integrate compiler inlining with our instrumentation to avoid the no-inlining cost, and we need to further understand timing discrepancies and calibration, along with inter-node timing considerations. Further visualizations can also be developed from the basic information we are collecting, and additional work is needed to coalesce the data into a network-level picture of application performance rather than a node-level picture.

REFERENCES